

ANALYZING "C" PROGRAMS FOR COMMON ERRORS

by

Dennis M. Frederick

B.S., University of Missouri, 1970

A REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

COMPUTER SCIENCE

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Approved by:


Dr. Gustafson

CHAPTER 1

INTRODUCTION

The cost of error detection and correction can be a significant contributor to the life cycle cost of software. Some researchers estimate that 40% of the life cycle cost of a software system is in the testing phase [WOLV84]. Clearly, detecting as many errors as possible as early as possible will lower the overall cost of a software development project.

The focus of this investigation is the development of an error detection tool for the C Programming Language which will contribute to a decrease in those costs for projects coded in C. This tool will detect errors involving logical and semantic errors in statements that are syntactically and semantically acceptable to the C compiler.

The C Programming Language has a number of pitfalls which can lead to errors. C is a powerful, high-level language with a rich repertoire of operators, data types and control flow constructs. However, the very richness that makes it attractive also results in a complexity that can be troublesome to both beginning and expert programmers alike. In my experience teaching C Language and consulting on C Language projects, I have noted several errors made frequently by beginners and occasionally by experienced programmers. These errors could be detected by the proposed tool.

There are already several types of tools available for error detection each embodying a slightly different approach to the problem. These include **Dynamic Trace** tools, **Core Dump Analysis** tools, **Static Analysis** tools, and **Goal Analysis** tools. An example of each of these types is explained in some detail in the paragraphs that follow.

Dynamic Trace tools like "ctrace" [UNIX84] allow step by step execution of C programs with outputs to show the values of important variables along the way. **Ctrace** takes the program source file as input. It inserts statements which will print the text of all executable statements and the values of all referenced variables. The output is then directed to standard output which is normally redirected to a temporary file. The C compiler is then invoked and the newly inserted print statements are compiled as part of the program. Upon execution, the text of each executable statement is printed as it is executed. The values of any variables referenced by the statement being executed are also printed.

When **ctrace** detects loops in a program, the loop is traced once, then tracing is stopped until the loop is exited or a different sequence of statements within the loop is executed. A warning message is printed every 1000 times through a loop to help detect infinite loops. Tools of this type are particularly useful for uncovering problems with control flow.

Core Dump Analysis tools like "sdb" [KATS81] are useful for examining "core

image" files produced by aborted program executions. **Sdb** (currently implemented for C and Fortran 77) has a variety of features and capabilities which make it valuable. When using **sdb** with a C program, one must compile the source with the "-g" option. **Sdb** can then be invoked after program execution has aborted and produced a "core dump". It will reveal the procedure name and line number in which the error occurred. Upon request **sdb** will also output a stack trace that consists of a list of the called procedures which led to the error. Included in that listing are the values of the input parameters at the time of the error. **Sdb** also has extensive capabilities for displaying variables of the program. Values of the variables can be displayed in a variety of user-selected formats and variable addresses are readily available upon request. Another useful feature of **sdb** is its ability to do breakpoint debugging. Once **sdb** is invoked, breakpoints can be set at any point in the program. Execution is initiated and continues until just before the breakpoint is reached. **Sdb** then halts the program and gives access to the core image examination capabilities already mentioned. Rounding out **sdb's** capabilities is the ability to single step the program and examine the core image after each statement. All these features together make **sdb** a very useful debugging tool.

Static Analysis tools like "lint" [JOHN81] examine program source files for a variety of possible misuses of the language. **Lint** pursues many types of errors which may lead to improper execution even though the program may compile

with no errors. Several of these are discussed below. **Lint** will issue warnings about variables and functions which are declared but never again referred to in the program. While these variables are usually left over from previous versions of the program and are probably harmless, they are flagged in the interest of encouraging good programming practice. Also, **lint** attempts to detect variables which are used in a program before being assigned a value. **Lint** also attempts to analyze the control flow of a program. It will complain about portions of a program which are apparently unreachable and loops which cannot be entered from the top or exited from the bottom. Finally, **lint** discourages older forms of the compound assignment operators as well as what are referred to as "strange constructions" such as tests that can never succeed (or fail) and statements that have no effect whatsoever.

Goal Analysis tools like "**Proust**" [SOLL85] (a debugging aid for Pascal) are useful in a teaching environment where specific programming assignments are given and the student's trial solution is analyzed for its effectiveness in solving the problem. **Proust** uses an interesting strategy for uncovering bugs in the proposed solution. First, it retrieves a description of the particular problem assignment from a library of such descriptions written in **Proust's** own problem-description language. The problem description is a paraphrase of the English language problem assignment given to the student. The problem description contains descriptions of each of the sub-problems or tasks that must

be performed successfully for the overall problem to be solved. Next, **Proust** draws from a library of valid solutions to each task, attempting to find a match with the student's approach to that task. If a match is found, **Proust** infers that the task is implemented correctly. If a match is not found, **Proust** checks a database of possible bugs to see if it can explain the discrepancies. It then reports an English language description of the bug and will sometimes go so far as to suggest input data that will demonstrate the presence of the bug. Processing continues in this manner until all the tasks of the problem have been analyzed.

All these tools are valuable members of the family of software development tools. However, none of them addresses the particular errors I have targeted for this investigation.

Errors of this class occur in statements that are syntactically and semantically correct. The errors escape detection by the C compiler and can be very difficult to find. Nonetheless, they are usually abuses of the language resulting in statements that probably do not accurately reflect the programmers intentions and will almost always cause execution errors. In limited cases these "errors of intent" do not cause execution errors, but reflect such bad programming practice and make a program so vulnerable to failure over its life cycle that they should be reported as errors. The specific errors under

consideration in this investigation involve misuse of the assignment and comparison operators, unintentional use of the null statement, errors of omission in switch/case statements, improper parameter specification in scanf() function calls and the use of uninitialized pointers. There may be other errors of this class equally worthy of detection effort.

CHAPTER 2

REQUIREMENTS

In order to find these "errors of intent" the tool must have several general capabilities. It must be able to identify those constructs which have the potential for harboring the errors being sought. Also, it must be able to filter out and discard statements not of interest to the tool. The tool needs the ability to make judgments concerning the appropriateness of constructs used in the suspect statements. These judgments will vary depending on the particular type of error being pursued at any given time. Finally it must be able to report the errors detected and the line numbers on which they were found.

The following paragraphs contain descriptions of the particular errors targeted by this tool along with more information concerning the capabilities required for the detection of each error.

ASSIGNMENT VS COMPARISON

One of the most common errors arises from confusion over the assignment and equality comparison operators. Some high level languages (notably Pascal and Ada) use `:=` for assignment and `=` for comparison, whereas, C Language uses

= for assignment and == for comparison. This inevitably leads to misuse of these operators by programmers already familiar with the other languages. Even programmers not already familiar with using the equal sign for comparison, find it a natural choice for that use. Furthermore, from my experience in the classroom, this error is very difficult for individuals to detect in their own programs.

For example, if a programmer has momentarily forgotten the operator definitions, a construct like:

```
if ( z = 4 )  
{  
.  
.  
.  
}
```

or

```
while ( x = y )  
{  
.  
.  
.  
}
```

can appear so reasonable and correct that programmers will almost always look elsewhere for an error. One can conceive of instances where a programmer might intentionally control a **while** loop with the assignment of one variable to another knowing the result will eventually be zero (false) thereby causing the loop to terminate. However, this is considered bad programming practice and

will be reported as a possible error by the proposed tool.

There are, of course, some legitimate uses of the assignment operator within control flow constructs. For example:

```
while ( (c = getchar()) != EOF )
{
    .
    .
    .
}
```

which is a very useful technique for assigning a character from standard input to a variable and then comparing it with the end of file marker in the same statement. This tool will not flag legitimate constructs such as this as errors.

Inappropriate use of the assignment operator is likely to occur within the test/comparison portion of the control flow constructs (**if**, **while**, **for**, **do-while**) therefore, my tool will analyze those statements for this class of errors. The tool must have the ability to distinguish between appropriate and inappropriate assignments and issue warnings only after encountering the latter.

THE NULL STATEMENT (;)

Another very frequent error is the unintentional termination of a control flow construct with the null statement (;). Most lines of code in a C Language program end with a semicolon. However, the control flow constructs (**if**, **while**,

and for) do not. If one places a semicolon after one of these constructs the behavior of the program can be altered significantly. For instance:

```
for ( i = 0; i < 10; i++)  
{  
    name[i] = tempname[i];  
}  
.  
.  
.
```

will copy the first ten elements of the **tempname** array to the **name** array, whereas:

```
for ( i = 0; i < 10; i++);  
{  
    name[i] = tempname[i];  
}  
.  
.  
.
```

will execute the null statement ten times, then copy the eleventh element of the **tempname** array to the **name** array.

There are legitimate uses of the null statement following control constructs as in:

```
while ( (c = getchar()) <= ' ');
```

which is a good technique for skipping over white space in an input line. However, a less confusing way to code this would be:

```
while ( (c = getchar()) <= ' ' )  
;
```

which shows more explicitly that the null statement is intended as the object of the **while** statement.

This tool will check for a new line between the control flow construct and the null statement and issue a warning if one does not appear.

OMISSION OF BREAK STATEMENTS

Another common error involves the omission of **break** statements in the cases of a **switch** statement. C handles this type of control flow construct differently from other languages in that execution proceeds from one **case** to the next until a **break** statement or the end of the **switch** is encountered. Programmers familiar with other languages will often forget to include **break** statements where they are needed, thereby producing a control flow different from that intended.

Consider the following code fragment for processing command line options and setting flags based on the options received. Assume the **-h** option has been entered on the command line. The statements associated with **case 'h'** will be executed.

```

while (argc > 1 && (*++argv)[0] != '-') {
    for (s = argv[0]+1; *s != ' '; s++) {
        switch (*s) {
            case 'h':
                help = TRUE; /* provide usage info */
                break;
            case 'f':
                file = TRUE; /* log output in a file */
                break;
            default:
                printf("Invalid Option %c0, *s);
                argc = 0;
                break;
        }
    }
    argc--;
}

```

If the **break** statement in case 'h' is omitted, execution will continue with the statements of case 'f'. Therefore, both the file and help flags will be set whenever the **-h** option is selected on the command line.

There are times when omission of **break** statements is intentional and the manner in which C handles cases becomes a convenient way to accomplish a logical ORing of case matches. This is illustrated in the following example.

```

switch(x) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        vowel++;
        break;
    default:
        consonant++;
        break;
}

```

This code fragment increments the vowel counter when either an **a**, **e**, **i**, **o**, or **u** is contained in **x**. My error checker will not issue warnings about constructs like this. It will assume the programmer intentionally omitted the **break** statements in order to create a logical OR structure in the switch. All other occurrences of missing **break** statements will be flagged as errors.

SCANF() FUNCTION CALLS

Yet another common error is improper specification of the input parameters in **scanf()** function calls. The **scanf()** function assumes that each of its input parameters is a valid address for storage of the information being scanned in. However, there are many ways to specify an address in C Language. Therefore, it is not surprising that confusion over the proper usage often leads to syntactically correct but numerically erroneous address specifications. Consider the following example.

```
main()
{
    char first[15];
    char last[15];
    char job[7];
    char *p;
    float sal;
    p = last;
    scanf("%s %s %c %d",first, p, &job[0], &sal);
}
```

In this example, `first`, `p`, `&job[0]`, and `&sal`, are all valid address references.

However, in the slightly modified version below, `first[0]`, `p`, `job`, and `sal` are all syntactically acceptable while only the reference to `job` would produce the intended result.

```
main()
{
    char first[15];
    char last[15];
    char job[7];
    char *p;
    float sal;
    p = last;
    scanf("%s %s %c %d",first[0], *p, job, sal);
}
```

To uncover problems of this nature the tool must produce a symbol table of all names used in the program. The table contains an indication of whether the name is an array, a pointer, or a single-element variable. Then all `scanf()` function calls are analyzed to determine whether the parameters are specified correctly. And, of course, the tool issues a warning message upon encountering an erroneous specification.

POINTERS

Another error that is frequently made in C Language involves the use of pointers which have not been initialized or which have been initialized incorrectly. In the following example the pointer **p** has been declared but never initialized so its use in the `scanf()` function call will produce unpredictable results.

```
main()
{
    char aircraft[10];
    char *p;
    scanf("%s", p );
}
```

And in the example below the pointer **p** has been initialized improperly (i.e. initialization should be `p = &x;`).

```
main()
{
    int x;
    int *p;
    p = x;
    scanf("%d", p );
}
```

My tool checks to see that each pointer declared is assigned a valid address before it is used in the program.

These then are the errors the debugging tool will attempt to uncover. The statements containing them are syntactically and semantically acceptable to the

compiler. Therefore, it must go beyond the level of checking for illegal syntax and semantics. It must be capable of assessing and making judgments about the programmers intentions, and issuing warnings if the constructs in question are suspected of harboring errors. And like all tools of this type it must be careful to complain only when there is a high probability that an error has actually occurred, lest it gain a reputation for issuing unnecessary warnings.

CHAPTER 3

DESIGN

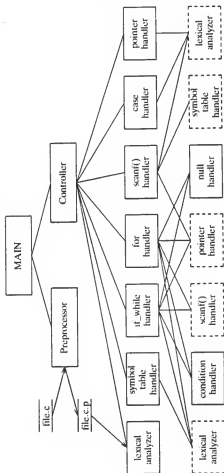
The tool I propose to uncover these errors is called **dust**. To identify these errors, **dust** must scan the source code, extract information about variables, ignore correct statements, and further investigate suspect statements. To accomplish this, **dust** consists of a preprocessor, a controller, a lexical analyzer, a symbol table handler, and a separate module for each error type being detected. The modules directly involved in error detection are: the `if_while` handler, the `for` handler, the condition handler, the null handler, the case handler, the `scanf()` handler, and the pointer handler. Overall organization of the program is shown in the hierarchy chart in Figure 3-1.

THE PREPROCESSOR

The purpose of this module is to strip out **#include** statements and comments from the source file and to perform the substitutions called for by the user's **#define** statements. The reason for deleting the **#include** statements is to limit the scope of the code being diagnosed to the source file.

The preprocessor works in the following manner. First, it takes the source file as input and executes a system command that removes the **#includes**,

Figure 3-1: Module Hierarchy



placing the output in a temporary file called "file.c.x". Next, it invokes the C preprocessor which draws its input from the temporary file, performs the **#define** substitutions and strips out the comments. The output is placed in another file called "file.c.p" and the temporary file "file.c.x" is removed. At this point the file "file.c.p" is ready for input to the lexical analyzer.

THE CONTROLLER

The next module invoked is the controller. This module directs the activities of all the other modules. It reads tokens from the input delivered by the lexical analyzer until it detects a token of interest to one of the other modules. If the token indicates the beginning of a statement related to one of the error types, control is passed to the appropriate error detection module. If the token indicates a declaration, control is passed to the symbol table handler. When control is passed to one of the modules below the controller, that module assumes the task of calling the lexical analyzer to obtain additional tokens as needed. When the task performed by the error detection module or symbol handler is completed, control is returned to the controller.

One special purpose task is built into the controller to handle the special problem presented by **do-while** statements. The closing **while** of a **do-while** statement is always terminated by a semicolon. If special steps were not taken,

this could cause the tool to produce erroneous null statement error messages.

Therefore, the controller keeps track of any **do-while** statements which have been opened by a **do** statement, but not as yet closed off by the corresponding **while** statement. This information is made available to other modules through a global variable. In this way, if a semicolon-terminated **while** statement is encountered, the null handler will know whether it is a null statement error, or merely the expected closing **while** of a **do-while** statement.

The tool, because of its modularity, is easy to expand. Additional error checks could be very easily incorporated into the tool by adding more error detection modules under the controller.

THE LEXICAL ANALYZER

This module has two very simple responsibilities. Its primary task is to break the source code into tokens. A secondary task is to keep track of the current line number.

It obtains input from "file.c.p" produced by the preprocessor. Output consists of a numeric token and a type indicator (i.e. keyword, constant, identifier or operator) for each call to the module.

THE SYMBOL TABLE HANDLER

This module builds a symbol table and answers queries from the other modules about entries in the table. Symbol table entries for each identifier will include the name, an assigned identifier number, variable type (i.e. array, structure, single-element, pointer, etc.) and, in the case of pointers, an indication of whether it has been assigned a value. The formation of this table is crucial to the operation of the error detection modules which rely on information about the identifiers.

THE IF_WHILE HANDLER

This module handles the error checking of **if** and **while** statements. Execution is triggered when the controller detects the presence of an **if** or **while** keyword. The module then brings in additional tokens from the lexical analyzer to complete the statement. Then it breaks the statement into its component parts: namely, the keyword segment, the conditional (e.g. $(x == y)$, $(a <= b)$, $((c = \text{getchar}()) \neq \text{EOF})$) and the object statement of the construct (if, and only if, it appears on the same line). Then it passes the conditional segment to the condition handler, and the object statement to the null handler. Also, if this module detects the presence of a **scanf()** function call, it calls the **scanf()** handler. The **scanf()** handler performs its error checking and then returns

control to the `if_while` handler to continue analyzing the statement. In addition, if it detects a pointer it calls the pointer handler which performs its error check and returns control to the `if_while` handler.

Input to this module consists of tokens from the lexical analyzer. Output is in the form of pointers to the conditional and object segments of the statement.

THE FOR HANDLER

This module handles the error checking of `for` statements. Execution is triggered when the controller detects the presence of a `for` keyword. The module then brings in additional tokens from the lexical analyzer to complete the statement. Like the `if_while` handler, it then breaks the statement into its component parts: namely, the keyword segment, the conditional (e.g. `(x == y), (a <= b), ((c=getchar()) != EOF)`) and the object statement of the construct (if, and only if, it appears on the same line). Then it passes the conditional segment to the condition handler, and the object statement to the null handler. Like the `if_while` handler it calls the `scanf()` handler when it detects a `scanf()` function call, and the pointer handler when it finds a pointer. Input to this module consists of tokens from the lexical analyzer. Output is in the form of pointers to the conditional and object segments of the statement.

THE CONDITION HANDLER

This module detects assignment operators appearing in the conditional construct that were really intended to be relational operators. As a by-product of this error checking, the module will also have some knowledge of possible operator precedence errors. Since it is convenient, these errors will be reported in addition to misuses of the assignment operator.

The condition handler performs a series of steps in pursuit of errors. First, it does a quick scan through the conditional construct to see if there are any assignment or relational operators present. If there are no assignment operators and no relational operators, error message 3 will be issued. If there is an assignment operator but no relational operator, the module assumes the assignment operator was intended to be a relational operator (most likely the equality relation) and, therefore, issues error message 1. If a relational operator is found, an additional check is performed. Specifically, the module checks to make sure the assignment operator has been forced to a lower precedence than the relational operator by the proper use of parentheses. If it has not, error message 2 is issued.

Input to this module is a pointer to the conditional segment of the control construct. Output is in the form of three possible error messages generated under the conditions described above. They are:

Error message 1: "line #: - Misuse of assignment operator (=) in "if" (while, for) - try (= =)".

Error message 2: "line #: - Operator precedence error involving assignment in "if" (while, for)".

Error message 3: "line #: - No relational operators in "if" (while, for)".

THE NULL HANDLER

This module determines whether a control construct has been terminated with a null statement on the same line of code. It receives as input the character string beginning with the first character after the closing parenthesis of the conditional construct and ending with the new line character. If it finds only a semicolon (possibly surrounded by spaces or tabs) followed by a new line, an error message will be output warning that a possible misuse of the null statement has occurred. A non-null statement followed by a semicolon will be considered acceptable on the same line of code, although to the purist this might also be considered bad programming practice.

Input to this module is a pointer to the object statement associated with the conditional. Output, in the event of an error, is the warning: "line #: - Null statement (;) after "if" (while, for)".

THE SCANF() HANDLER

This module is responsible for analyzing the arguments to `scanf()` function calls to determine whether they provide valid address specifications. It will acquire the variable type from the symbol table for each identifier in the function call. Then it will decide whether the syntax used in the argument list is appropriate. If the syntax is not appropriate it will issue a warning message.

This module receives as input a pointer to the beginning of the `scanf()` argument list. Output upon detection of an error is the warning: "line #: - Incorrect address specification for "variable" in `scanf()`".

THE CASE HANDLER

This module reviews the statements associated with the `case` labels of `switch/case` statements to insure that a `break` statement is among them. If one or more `cases` do not contain `breaks` the module will issue a warning message. An exception to this case is when two or more `case` labels appear sequentially with no statements in between. This is a convenient way to form an "or" construct in this language and will not be flagged as an error. However, the last `case` in the sequence is still required to have a `break` statement associated with it.

Input to this module is a pointer to the first **case** label in the **switch** statement. Output on detection of an error is the message: "line #: - No break at end of "case".

THE POINTER HANDLER

This module keeps track of pointer initializations, and "first uses" of each pointer. The pointer handler will update the symbol table when a pointer is first initialized. Then later in the program when the pointer appears again, it will check to verify that it has been initialized. If the pointer is used without first being assigned a value, the module issues an error message. Input to the module is the identity of the pointer variable in question. Output in the event of an error is the message: "line #: - Possible uninitialized pointer - "variable".

THE USER INTERFACE

This section describes how the user executes the tool, how options are specified, and how certain features of the tool will help a novice user. The name of the error detection tool is "**dust**". The simplest form of the command line calling for the application of **dust** to a source file would consist of the name **dust** followed by the source file specification.

Example:

```
$ dust program.c
```

By default all error checks are activated. The user can selectively suppress any of the checks by selecting the appropriate command line option. The options and their meanings are described below.

Options:

-h	HELP! Print usage information.
-a	Suppress check for inappropriate assignment.
-b	Suppress check for break statements
-n	Suppress check for null statements.
-p	Suppress check for uninitialized pointers.
-s	Suppress check of scanf() function arguments.

Multiple options can be placed behind a single dash or each can be given its own dash prefix. Furthermore, the order in which the options appear on the command line is unimportant as long as all options appear before the source file is specified. In the following examples the "-a" and "-n" options are selected to suppress the checks for inappropriate assignments and unintentional null statements. They are all equivalent usages.

Examples:

```
$ dust -an program.c
$ dust -na program.c
$ dust -a -n program.c
$ dust -n -a program.c
```

Selecting the "-h" option will cause a short help message to be printed on the terminal. The help message includes a description of the tool along with a list of all the available options and their meanings.

The user interface will issue error messages about invalid options, no source file specification, or a source file specification that does not end in ".c".

CHAPTER 4

IMPLEMENTATION

Dust was implemented using approximately 1600 lines of C - Language code. Each of the error checks is performed by a separate function called from the mainline program, or in some cases called from another error checking routine. Details of the code can be found in the appendices.

Several of the author's C programs were run through **dust** with very encouraging results. In some cases, errors were deliberately planted in the programs to test the tool's effectiveness. **Dust** consistently detected and reported all the errors it was designed to uncover and did not report errors where none existed.

Average CPU times were computed for programs of various sizes. Overall, **dust** consumed 1.92 seconds per 100 lines-of-code processed running on a Digital Equipment Corporation VAX 11/780. Large programs tended to have a lower time to lines-of-code ratio than small ones.

Several example runs of **dust** appear below. The sample program used here was constructed specifically to demonstrate some of the error checking capabilities of **dust** - it performs no real computing function. Some features of the user interface are also demonstrated by intentional omission or faulty entry

of command line arguments.

```
$ pr -tn test3.c
1  main()
2  {
3  int x = 2, y = 3;
4  if ( x = y );
5  while ( x | y )
6  ;
7  while ( x >= y );
8  for (x = 0; x = y;x++ );
9  }
```

\$ dust

You must specify a source file to be checked!

For help use: dust -h

\$ dust -h

This command searches C Language programs for a variety of errors.

It assumes your program has compiled successfully, but is not running properly. By default, all the error checks are activated. To selectively suppress any of the checks, use the appropriate command line option(s).

OPTIONS:

- h HELP!
- a Suppress check for inappropriate assignments.
- b Suppress check for breaks in switch/case statements.
- n Suppress check for unintentional null statements.
- p Suppress check for uninitialized pointers.
- s Suppress check for improper scanf() function arguments.

EXAMPLES:

```
$ dust program.c
```

```
$ dust -an program.c
```

```
$ dust test3
```

Source file name must end with ".c"

```
$ dust test5.c
```

Can't open "test5.c" for reading.

```
$ dust test3.c
```

line 4: - Misuse of assignment operator (=) in "if" (try ==).

line 4: - Null statement (;) after "if".
line 5: - No relational operators in "while".
line 7: - Null statement (;) after "while".
line 8: - Misuse of assignment operator (=) in "for" (try < =).
line 8: - Null statement (;) after "for".

\$ dust -z test3.c
Invalid Option z
For help use: dust -h

\$ dust -n test3.c
line 4: - Misuse of assignment operator (=) in "if" (try = =).
line 5: - No relational operators in "while".
line 8: - Misuse of assignment operator (=) in "for" (try < =).

\$

CHAPTER 5

CONCLUSION

The debugging tool **dust** is a useful addition to the family of software development tools for C - Language. **Dust** finds errors in usage of the language commonly made by beginning programmers and occasionally made by experienced programmers. The errors it finds are typically difficult to detect without the aid of such a tool. The tool does not report errors where none exist, and it consistently finds all the errors it was designed to uncover.

The tool could be extended to include some additional checks not included in this design due to time constraints. One addition could be a check of the arguments to `printf()` function calls (similar to the `scanf()` check already implemented). Another could be a check for the presence of nested comments which are illegal in C, but which are detected by the compiler only indirectly. Another useful check would be a simple check for the presence of a semicolon at the end of each statement (ignoring, of course, those statements which should *not* end in a semicolon). The absence of a semicolon where it is needed usually generates a long list of compiler syntax errors which do not always pinpoint the line of code where the semicolon has been omitted. An explicit check of this kind would save time and should be fairly easy to implement.

BIBLIOGRAPHY

1. [SOLL85] "PROUST: An Automatic Debugger for Pascal Programs", Elliot Solloway and W. Lewis Johnson, Byte Magazine April, 1985 pg179-190.
2. [UNIX84] "UNIX System V User Reference Manual (Release 2.0)", AT&T Bell Laboratories, 1984.
3. [WOLV84] "Software Costing", R. W. Wolveton from Handbook of Software Engineering edited by C. R. Vick and C. V. Ramamoorthy: Von Nostrand Reinhold Co. Inc, New York 1984.
4. [JOHN81] "Lint, a C Program Checker". S. C. Johnson from Documents for Unix Vol 1, January 1981, Bell Telephone Laboratories, Inc.
5. [KATS81] "SDB - A Symbolic Debugger", H. P. Katseff from Documents for Unix Vol 1, January 1981, Bell Telephone Laboratories, Inc.

APPENDIX A

USER'S MANUAL

DUST

DUST

NAME

dust - check for common C programming errors

SYNOPSIS

```
dust [ -habnps ] file.c
```

DESCRIPTION

Dust is a debugging tool for C - Language programs. It searches for several common abuses of the language which are not detected by the compiler. Specifically, it detects inappropriate uses of the assignment operator and operator precedence errors in control constructs, unintended null statements, omission of break statements from switch/case constructs, improper address specifications in scanf() function calls, and uninitialized pointers. It does not do the normal syntax and semantics checking done by the compiler. In fact, *dust* assumes the program being tested has already compiled successfully, but is not operating correctly.

By default, all the error checks are activated. However, the user can suppress checks by selecting the appropriate option(s) below.

- | | |
|----|---|
| -h | HELP! Print usage information. |
| -a | Suppress check for inappropriate assignments. |
| -b | Suppress check for break statements. |
| -n | Suppress check for null statements. |
| -p | Suppress check for uninitialized pointers. |
| -s | Suppress check of scanf() function arguments. |

APPENDIX B

FUNCTION DESCRIPTIONS

This appendix contains detailed descriptions of each function in the program. Descriptions include the purpose of the function, the inputs and outputs, and an explanation of the more important features of the code itself.

`lex()`

Purpose:

The lexical analyzer (`lex()`) reads source code from the file "file.c.p" produced by the preprocessor and forms the characters into the tokens of the language (keywords, identifiers, operators etc.).

The `lex()` function calls several subordinate functions to form tokens while some are formed within `lex()` itself. Tokens consisting of a single character are formed by `lex()`, with the exception of single character identifiers which are formed by `key_id()`. Tokens comprised of more than one character are formed by functions subordinate to `lex()`. As soon as the first character of the token is brought in by `getch()`, a decision is made as to which function will process the remaining characters. If the first character is a letter or underscore, control is passed to the `key_id()` function. If the first character is not a letter, control passes to the function associated with the case matched by that character. (See the large switch/case statement in `lex()`).

When the EOF is encountered, `lex()` takes care of closing and removing the working input file (`filename.c.p`) and exiting the command. The saves all the other functions that call `lex()` from the trouble of checking each time to see if EOF has been encountered.

`Lex()` keeps track of the value and type of the last token (in `Lastok` and `Lastype`). Also, it keeps track of special tokens which are of interest to the `case_hand` function (in `Lastcasetok` and `Lastcasetype`). Namely, the last token which was not a semicolon and not a newline. The `case_hand` function uses this information when making its decision about whether a break statement has been omitted from a case.

Input:

Characters are read in using calls to the `getch()` function which returns a single character from the input file each time it is called. Characters can also be "put back" on the input stream, by calls to the `ungetch(c)` function. Successive calls to `getch()` will get the characters that were "put back" before getting new characters from the file.

Output:

The external variables Token and Type are assigned values by lex(). If the token is a keyword or operator, Token is assigned the value given by the #define statement for that operator or keyword. If the token is a constant, Token is given the value of the constant. (Note: lex() does not attempt to form floating point, or character constants into single tokens. For example, a floating point constant would be broken into three tokens. Namely, the integer part, the decimal point, and the fractional part.) If the token is an identifier, lex() calls a hashing function which converts the identifier character string to an integer between 0 and 2000. That integer is then assigned to Token. Collisions (i.e. different character strings hashing to the same value) are handled by the symbol table handler to make sure that each identifier gets its own entry in the table. Type is assigned one of the following values: 'o' if the token is an operator, and 'c' if its a constant.

Important variables:

Token integer value for token.

Type char value for type of token.

Lastok integer value for previous token.

Lasttype char value for type of previous token.

Lastcasetok integer value for previous token of interest to case handler.

Lastcasetype ... char value for type of previous token of interest to case handler.

Charbuff[] char array for storage of input characters as token is being formed.

Charpos integer index into Charbuff[] array.

Other internal functions called:

- what_type()
- key_id()
- numproc()
- exclamproc()
- percentproc()
- amperproc()
- starproc()
- plusproc()
- minusproc()
- slashproc()
- lessproc()
- equalproc()
- greatproc()
- vorproc()
- pipeproc()
- getch()

key_id()

Purpose:

The purpose of `key_id()` is to determine whether the character string it processes is a keyword or an identifier and then assign the appropriate value to `Token` and `Type`. It is called whenever the first character of a new token is a letter or underscore. The first thing it does is bring in the remainder of the string (all letters, digits or underscores) and put them in the `Charbuff[]` array. It then `ungetch()`'s the last character `getch()`'ed (since we know it is not part of this token - i.e. not a letter, digit, or underscore) and null terminates the string in `Charbuff[]`. It then resets `Charpos` to point to the beginning of the string and calls `findkey()` to see if the token is a keyword. If it is a keyword, `Type` gets the value `KEYWD` and `Token` gets the value `keytable[n].keynum` (`n` = return value of `findkey`). The structure `keytable` is a table consisting of character strings which are the keywords, and their corresponding numeric values (defined by `#define` statements). If the token is not a keyword, `Type` gets the value `ID` and `Token` gets an integer value representing the character string returned by the function `pchash()`.

Input:

Characters are brought in by calls to `getch()` and placed in the character array `Charbuff[]`.

Output:

`Type` is assigned the value `KEYWD` if the token is a keyword. `Type` is assigned the value `ID` if the token is an identifier. `Token` is assigned the value corresponding to the keyword found (`keytable[n].keynum`) or an integer value returned by `pchash()` representing the identifier.

Important variables:

`Token` integer value for token.

`Type` char value for type of token.

`Charbuff[]` char array for storage of input characters as token is being formed.

`Charpos` integer index into `Charbuff[]` array.

`keytable[n].keynum` integer value defined for the keyword of index `n`.

`n` holds value returned from `findkey` (the index into the keyword table).

Other internal functions called:

what_type()
getch()
ungetch()
findkey()
pchash()

findkey()

Purpose:

The findkey() function does a binary search through keytable (indirectly accessed by local structure tab) to find the keyword (if any) which matches the string in Charbuff[] (accessed through local variable word). This search uses the strcmp() function for string comparisons. If the first string is lexically "less than" the second, a negative value is returned. If the first string is lexically "greater than" the second, a positive value is returned. If the first string is "equal to" the second, zero is returned. The search progresses by setting the high, low, and mid values depending on the returns from strcmp(). Therefore, the strings in the lookup table must be in ascending alphabetic order for this binary search algorithm to work.

Input:

Input parameters are a pointer to Charbuff[], a copy of the structure keys, and the size of the keytable.

Output:

Returns the value of the index into keytable for the keyword matched, or the value -1 if a keyword match was not found (in which case the token must be an identifier).

Important variables:

Charbuff[] char array for storage of input characters as token is being formed.

low low boundary of search

mid midpoint of search

high high boundary of search

cond integer that holds return value of strcmp().

Other internal functions called:

none

getch()

Purpose:

The purpose of getch() is to bring in a single character from the file associated with the FILE pointer inptr each time it is called (inptr points to filename.c.p). It first looks to see if there is a character in the buffer Buf (the result of an ungetch()), before going to the file for a character.

Input:

A single character from filename.c.p.

Output:

Returns the character gotten.

Important variables:

Buf[] external character array to buffer input characters.

Buip external index into the Buf[] array.

Other internal functions called:

none

ungetch()

Purpose:

The purpose of `ungetch()` is to "put a character back on the input stream". It actually puts the character back into the buffer `Bufl[]`, but this is transparent (and irrelevant) to the calling function. The next call to `getch()` gets the character "put back" by the previous `ungetch()` (if any).

Input:

The character to be "put back" is delivered to `ungetch()` as an input parameter.

Output:

One character to the buffer `Bufl[]`.

Important variables:

`Bufl[]` external character array to buffer input characters.

`Buflp` external index into the `Bufl[]` array.

`c` temporary storage for character to be "put back".

Other internal functions called:

none

what_type()

Purpose:

The purpose of what_type is to determine whether a character is a letter (a - z, A - Z, or the underscore) or a digit (0 - 9).

Input:

The character to be checked for type is input as a parameter.

Output:

Output is a return value of either LETTER or DIGIT.

Important variables:

c temporary storage for character to be "put back".

Other internal functions called:

none

numproc()

Purpose

The purpose of numproc() is to process tokens that consist entirely of digits. It is called whenever the first character of the token is a digit (0-9). In this case the token must be a number since identifiers cannot begin with a digit.

Input:

Characters are read in using calls to the getch() function which returns a single character from the input file each time it is called. Character are read into Charbuff[] until a non-digit is found. Then the non-digit is ungetch()'ed and the string in Charbuff[] is null terminated.

Output:

Type is assigned the value CONST. Token is assigned the actual numeric value of the token (returned by atoi(Charbuff) - ascii to integer conversion).

Important variables:

Token integer value for token.

Type char value for type of token.

Charbuff[] char array for storage of input characters as token is being formed.

Charpos integer index into Charbuff[] array.

Other internal functions called:

- getch()
- ungetch()

`exclamproc()` `percentproc()` `amperproc()` `starproc()` `plusproc()` `minusproc()` `slashproc()`
`lessproc()` `equalproc()` `greatproc()` `xorproc()` `pipeproc()`

Purpose:

The purpose of these functions is to process the operator tokens. The first character brought in by `lex()` determines which function will be called (see the large switch/case statement in `lex()`). The functions all consist of if/else if constructs which determine what the following characters are and therefore what the operator is. Any unwanted characters `getch()`'ed (i.e. characters which cannot be a part of any operator beginning with the first character already received) are `ungetch()`'ed immediately and the string in `Charbuff[]` is null terminated.

Input:

Characters are read in using calls to the `getch()` function which returns a single character from the input file each time it is called. Characters are read into `Charbuff[]` as the token is formed.

Output:

Type is assigned the value OP. Token is assigned the value for the operator as specified in the `#define` statements.

Important variables:

Token integer value for token.

Type char value for type of token.

`Charbuff[]` char array for storage of input characters as token is being formed.

`Charpos` integer index into `Charbuff[]` array.

`c` temporary storage for character to be "put back".

Other internal functions called:

`getch()`
`ungetch()`

`symhand()`

Purpose:

The purpose of the `symhand` function is to determine the variable type (single element variable, array, pointer, etc.) and call the `make_entry()` function to make the actual table entry.

Input:

Tokens are brought in using calls to the `lex()` function which returns a single token from the input file each time it is called.

Output:

none

Important variables:

`Token` integer value for token.

`token` local integer value for token.

`vartype` type of variable

`init` initialize flag

Other internal functions called:

`make_entry()`

make_entry()

Purpose:

The purpose of the make_entry function is to make entries in the symbol table. Each entry includes the token value, the variable name, the type of variable, the block number in which the variable is declared, and the initialized flag (YES or NO).

Input:

vartype, token, name, and init from calling function

Output:

Important variables:

token local integer for token value

vartype type of variable

name name of variable

init local initialize flag

Symtable[.newtoken Symbol table entry for token value

Symtable[.id .. Symbol table entry for variable name

Symtable[.block Symbol table entry for code block

Symtable[.vartype Symbol table entry for variable type

Symtable[.init Symbol table entry for init flag

Other internal functions called:

none

find_entry()

Purpose:

The purpose of the find_entry function is to find entries in the symbol table. Information drawn from the symbol table on the "searched for" variable includes the token value, the variable name, the type of variable, the block number in which the variable is declared, and the initialized flag (YES or NO).

Input:

token from calling function

Output:

Passes back information on variable to calling function.

Important variables:

token local integer for token value

vartype type of variable

name name of variable

init local initialize flag

Symtable[.].newtoken Symbol table entry for token value

Symtable[.].id .. Symbol table entry for variable name

Symtable[.].block Symbol table entry for code block

Symtable[.].vartype Symbol table entry for variable type

Symtable[.].init Symbol table entry for init flag

Other internal functions called:

none

`if_while_hand()`

Purpose:

The purpose of the `if_while_hand` is to process statements that begin with the `if`, `while`, or `do` keywords. Processing for `do` statements consists of pushing the `do` onto a stack (`Dostack`) to be popped later by the corresponding `while` of the `do/while`. This is to prevent the closing `while`'s of `do/while` statements from being mistakenly flagged as "null statement following `while`" errors. Processing of `if` and `while` statements consists of stripping out the conditional segment of the statement (the test between the outermost parentheses) and placing the tokens in the local buffer `tokbuf[]` and `typebuf[]` for the token values and types respectively. The function expects the next character after the `if` or `while` to be an open parenthesis. When it encounters this it increments `parentnt`, then enters a while loop which brings in the rest of the characters out to and including the closing parenthesis of the conditional. This is accomplished by incrementing `parentnt` for every open parenthesis and decrementing it for every close parenthesis. When `parentnt` reaches zero we have found the closing parenthesis of the conditional. It then calls `cond_hand()` to evaluate the conditional and `null_hand` to look for null statement errors (if the appropriate flags are turned on).

Input:

Tokens are brought in using calls to the `lex()` function which returns a single token from the input file each time it is called. Token values and types are read into the local buffers `tokbuf[]` and `typebuf[]`. Conditional type (`if`, `while`, `do`) is received as an input parameter to the function.

Output:

`tokbuf[]`, `typebuf[]`, and `condtype` are passed to the `cond_hand()` function. `condtype` is passed to the `null_hand` function.

Important variables:

`condtype` type of conditional (`if`, `while`, `do`).

`tokbuf[]` local buffer for tokens in conditional segment.

`typebuf[]` local buffer for token types in conditional segment.

`Dostack` stack for `do`'s and `{`'s - popped by `while`'s and `}`'s.

`Doptr` index into `Dostack`

`parentnt` counter for open and close parentheses.

Token integer value for token.

Type char value for type of token.

Assignments ... flag for activating check for misused assignments (cond_hand).

Null_stmts flag for activating check for null statements after if/while.

keytable[] to retrieve name corresponding to condtype

Other internal functions called:

- lex()
- cond_hand()
- null_hand()

for_hand()

Purpose:

The purpose of the for_handler is to process statements that begin with the for keyword. Processing consists of stripping out the conditional segment of the for (the test between the semicolons) and placing the tokens in the local buffers tokbuf[] and typebuf[] for for token values and token types respectively. The for_hand function brings in tokens after the for until it encounters the first semicolon and puts an open parenthesis into the token buffer in place of the it. Then it gets all tokens up to the next semicolon and puts them in the buffer. A closing parenthesis is then put in the buffer in place of the semicolon. The semicolons are replaced by open and close parenthesis so that the string delivered to cond_hand will look the same whether it came from an if, while, or for.

Input:

Tokens are brought in using calls to the lex() function which returns a single token from the input file each time it is called. Token values and types are read into the local buffers tokbuf[] and typebuf[]. Conditional type (for) is received as an input parameter to the function.

Output:

tokbuf[], typebuf[], and condtype are passed to the cond_hand() function. condtype is passed to the null_hand function.

Important variables:

condtype type of conditional (if, while, do).

tokbuf[] local buffer for tokens in conditional segment.

typebuf[] local buffer for token types in conditional segment.

Token integer value for token.

Type char value for type of token.

Assignments ... flag for activating check for misused assignments (cond_hand).

Null_stmts flag for activating check for null statements after if/while.

keytable[] to retrieve name corresponding to condtype

Other internal functions called:

- lex()
- cond_hand()
- null_hand()

`cond_hand()`

Purpose:

The purpose of the `cond_hand` function is to analyze the conditional (test) segment of `if`, `while`, and `for` statements. It first scans through the tokens in the conditional (using `tokbuff[]` and `typebuff[]` to count the number of assignment and conditional operators. (See first `switch/case` statement in function). It does not count assignment operators preceded by a single quote since these are character constants (`'='`). If there are no relational operators and no assignment operators then no test is being performed in the conditional and an error message is issued. If there are assignment operators but no conditionals, a misuse of the assignment operator has probably occurred (i.e. `=` instead of `==`) and a different error message is issued. If there are both assignments and relationals then the statement is checked for proper operator precedence (i.e. the assignment should be forced to higher precedence than the relational by the use of parentheses). (Refer to second `switch/case` in function).

If an assignment is encountered, all the tokens are read up to and including the next relational operator. An open parentheses decrements `parentcnt`. A close parentheses increments `parentcnt`. If further assignments are encountered, `assignment` is decremented. If after the relational operator is encountered, the `parentcnt` is less than an or equal to zero then we did not have an unmatched close parenthesis between the assignment and the relational operator. This means we have an operator precedence error.

If a relational operator is encountered, all the tokens are read up to and including the next assignment. An open parentheses increments `parentcnt`. A close parentheses decrements `parentcnt`. If after the assignment operator is encountered, the `parentcnt` is less than an or equal to zero then we did not have an unmatched open parenthesis between the relational and the assignment operator. This means we have an operator precedence error.

This process continues until either `assignment` or `relopcnt` goes to zero.

Input:

`tokbuff[]`, `typebuff[]`, and `condtype` are passed in from `if_while_hand` and `for_hand`.

Output:

Appropriate error messages if errors are detected.

Important variables:

`condtype` type of conditional (`if`, `while`, `do`).

`tokbuff[]` local buffer for tokens in conditional segment.

`typebuff[]` local buffer for token types in conditional segment.

assignment counter for assignment operators.

relopent counter for relational operators.

optype[] array for storing type of op (ASSIGN or RELOP)

Other internal functions called;

none

null_hand()

Purpose:

The purpose of the null_hand() function is to check for the presence of null statements on the same line as if, while, or for constructs. A null statement appearing on the line following the if, while, or for is acceptable. This function is called from the if_while_hand and the for_hand functions. First null_hand checks condtype to see if the construct is a for. If it is, we must first arrive at the closing parenthesis of the for. (If its an if or a while, we're already there). Therefore we call lex() as long as the token is not a close parenthesis. Once we arrive at the close parenthesis we make an additional call to lex() to move to the first token beyond the close parenthesis of the for. We then make additional calls to lex() until we encounter either a newline or semicolon, incrementing loopcnt each time through the loop. (A loopcnt value of zero means no statements were encountered before the newline or semicolon). If we encountered a newline we will increment the loopcnt. If we encountered a semicolon with no statement since the closing parenthesis of the for, we may have a null statement error. If condtype is while and we think we may have a null statement error, we must first check to see if there is a do on the Dostack. If there is not a do on the Dostack, we have a null statement error and we issue the appropriate error message. If there is a do on the Dostack then this null-terminated while is appropriate. We simply pop the do off the Dostack and continue. If condtype is either if or for and loopcnt is zero, we have a null statement error and we issue the appropriate error message.

Input:

condtype (if, while, for, do) from if_while_hand or for_hand.

Output:

Appropriate error messages if errors are detected.

Important variables:

condtype type of conditional (if, while, do).

Dostack stack for do's and { 's - popped by while's and } 's.

Doptr index into Dostack

Token integer value for token.

Type char value for type of token.

loopcnt counter for statements before newline or semicolon

keytable[] to retrieve name corresponding to condtype

Other internal functions called:

lex()

`case_hand()`

Purpose:

The purpose of the `case_hand` function is to check for missing breaks in switch/case statements. If the last non-newline, non-semicolon token (before the case token was encountered) was either a colon or a break or if the `switchflag` was on (meaning this is the first case in the switch) then there is no error. In this case we simply turn the `Switchflag` off (it may already be off, but that's okay). However, if the last non-newline, non-semicolon token was not a colon or break and the flag was off, we have a missing break statement in the switch. The function `lex()` as mentioned earlier, keeps track of the last tokens of interest to this function.

Input:

none

Output:

Appropriate error messages if errors are detected.

Important variables:

`Lastok` integer value for previous token.

`Lastype` char value for type of previous token.

`Switchflag` flag to indicate if this is the first case in switch.

Other internal functions called:

none

scanf_hand()

Purpose:

The purpose of the `scanf_hand` function is to check for incorrect address specifications in `scanf()` function calls. It checks the type of the variables in the `scanf()` argument list, (i.e. single variable, array, pointer, pointer array) and determines whether the syntax used to specify the address is appropriate. If not, a warning message is issued.

Input:

Tokens are brought in using calls to the `lex()` function which returns a single token from the input file each time it is called. Token values and types are read into the local buffers `tokbuf[]` and `typebuf[]`.

Output:

Appropriate error messages if errors are detected.

Important variables:

`Lastok` integer value for previous token.

`Lasttype` char value for type of previous token.

`Charbuff` character input buffer.

`Token` integer value for token.

`Type` char value for type of token.

`tokbuf` local token buffer

`typebuf` local token type buffer

`vartype` type of variable

`name` name of variable

Other internal functions called:

`find_entry()` is used to find variables in the symbol table.

point_hand()

Purpose:

The purpose of the point_hand function is to check for uninitialized pointers.

Input:

A token and a variable name are passed in from calling function.

Output:

Appropriate error messages if errors are detected.

Important variables:

token integer value for token.

vartype type of variable

name name of variable

init initialize flag

Other internal functions called:

find_entry()

`set_init()`

Purpose:

The purpose of the `set_init` function is to set the initialized flag associated with a pointer whenever that pointer is initialized. This flag can then be checked whenever needed to see if the pointer is initialized.

Input:

A token and a variable name are passed in from calling function.

Output:

none

Important variables:

`token` integer value for token.

`name` name of variable

`init` initialize flag

Other internal functions called:

`none()`

APPENDIX C

DUST SOURCE CODE LISTING

```

1  /*
2  * NAME: dust.c
3  * PROGRAMMER: Dennis Frederick
4  * DATE: June 7, 1987
5  *
6  *
7  *
8  *
9  *
10 * INPUTS: filename.c
11 *
12 *
13 *
14 *
15 *
16 *
17 * OUTPUTS: error messages
18 *
19 *
20 *
21 *
22 *
23 *
24 * OPTIONS:
25 *
26 * -h  HELP! Print usage information.
27 * -a  Suppress check for inappropriate assignment.
28 * -b  Suppress check for break statements in switch/case statements.
29 * -n  Suppress check for unintentional null statements.
30 * -p  Suppress check for uninitialized pointers.
31 * -s  Suppress check for improper scanf() function arguments.
32 *
33 *
34 *
35 *
36 *
37 */
38
39 /*
40 * DEFINES FOR C LANGUAGE KEYWORDS
41 */
42
43 #define ASM      0
44 #define AUTO     1
45 #define BREAK    2
46 #define CASE     3
47 #define CHAR     4
48 #define CONTINUE 5
49 #define DEFAULT  6
50 #define DO       7
51 #define DOUBLE   8
52 #define ELSE     9
53 #define ENTRY    10
54 #define ENUM     11
55 #define EXTERN   12
56 #define FLOAT    13

```

```

57      #define FOR          14
58      #define FORTRAN      15
59      #define FSCANF       16
60      #define GOTO         17
61      #define IF           18
62      #define INT          19
63      #define LONG         20
64      #define MAIN         21
65      #define REGISTER     22
66      #define RETURN       23
67      #define SCANF        24
68      #define SHORT        25
69      #define SIZEOF       26
70      #define SSCANF       27
71      #define STATIC       28
72      #define STRUCT       29
73      #define SWITCH       30
74      #define TYPEDEF      31
75      #define UNION        32
76      #define UNSIGNED     33
77      #define VOID         34
78      #define WHILE        35
79
80      /*
81      * DEFINES FOR INPUT TOKENS
82      */
83
84      #define NOT           100
85      #define NE            101
86      #define REM           102
87      #define REMEQ         103
88      #define AND           104
89      #define LOGAND        105
90      #define ANDEQ         106
91      #define OPENPAR       107
92      #define CAST          108
93      #define CLSPAR        109
94      #define STAR          110
95      #define STAREQ        111
96      #define PLUS          112
97      #define PPLUS         113
98      #define PLUSEQ        114
99      #define COMMA         115
100     #define MINUS          116
101     #define MMINUS         117
102     #define MINUSEQ        118
103     #define ARROW          119
104     #define DOT            120
105     #define SLASH          121
106     #define SLASHEQ        122
107     #define LT             123
108     #define SHFTL          124
109     #define SHFTLEQ        125
110     #define LE             126
111     #define EQ             127
112     #define EQEQ           128
113     #define GT             129
114     #define GE             130
115     #define SHFTR          131
116     #define SHFTREQ        132

```

```

117 #define QUEST 133
118 #define OPENBRAC 134
119 #define CLSBRAC 135
120 #define XOR 136
121 #define XOREO 137
122 #define OR 138
123 #define OREQ 139
124 #define LOGOR 140
125 #define TWOSCOMP 141
126 #define OPENBRACE 142
127 #define CLSBRACE 143
128 #define SEMI 144
129 #define COLON 145
130 #define DBLQT 146
131 #define SNGLOT 147
132 #define POUND 148
133 #define BKSLASH 149
134 #define NL 150
135 #define OPENCOMM 151
136 #define CLSCOMM 152
137 #define ATSIGN 153
138 #define GRAVE 154
139 #define DOLLAR 155
140
141 #define OP 'o' /* operator token id */
142 #define KEYWD 'k' /* keyword token id */
143 #define ID 'i' /* identifier token id */
144 #define CONST 'c' /* constant token id */
145
146 #define ASSIGN 'a' /* assignment operator id */
147 #define RELOP 'r' /* relational operator id */
148 #define LOGOP 'l' /* logical operator id */
149
150
151 #define BUFSIZE 100 /* input buffer size */
152 #define HASHSIZE 2000 /* hash table size */
153
154 #define LETTER 'a' /* letter id */
155 #define DIGIT '0' /* digit id */
156 #define NKEYS 36 /* number of keywords */
157
158 #define PTR 'p' /* pointer id */
159 #define PTRARRAY 'z' /* pointer array id */
160 #define ARRAY 'a' /* array id */
161 #define SINGLE 's' /* single variable id */
162
163 #define TRUE 1
164 #define YES 1
165 #define ON 1
166 #define FALSE 0
167 #define NO 0
168 #define OFF 0
169
170 #include <stdio.h>
171
172 /*
173  * EXTERNAL DECLARATIONS
174  */
175
176 struct keys { /* keyword lookup table */

```



```

177 char keyword[10];
178 int keyword;
179 } ktable[NKEYS] = {
180     "asm", ASM,
181     "auto", AUTO,
182     "break", BREAK,
183     "case", CASE,
184     "char", CHAR,
185     "continue", CONTINUE,
186     "default", DEFAULT,
187     "do", DO,
188     "double", DOUBLE,
189     "else", ELSE,
190     "entry", ENTRY,
191     "enum", ENUM,
192     "extern", EXTERN,
193     "float", FLOAT,
194     "for", FOR,
195     "fortran", FORTRAN,
196     "fscanf", FSCANF,
197     "goto", GOTO,
198     "if", IF,
199     "int", INT,
200     "long", LONG,
201     "main", MAIN,
202     "register", REGISTER,
203     "return", RETURN,
204     "scanf", SCANF,
205     "short", SHORT,
206     "sizeof", SIZEOF,
207     "sscanf", SSCANF,
208     "static", STATIC,
209     "struct", STRUCT,
210     "switch", SWITCH,
211     "typedef", TYPEDEF,
212     "union", UNION,
213     "unsigned", UNSIGNED,
214     "void", VOID,
215     "while", WHILE
216 };
217
218 char Buf[BUFSIZE]; /* input buffer */
219 char Type; /* type of token */
220 char Charbuf[BUFSIZE]; /* character input buffer */
221
222 int Bufp; /* Buf array index */
223 int Charpos; /* Charbuf array index */
224 int Token; /* input token */
225 int Linesum = 1; /* current line number */
226 int Block; /* current code block */
227 int Dostack[BUFSIZE]; /* tracks do statements */
228 int Doptp; /* Dostack array index */
229 int Lastok; /* previous input token */
230 int Lasttype; /* type of previous token */
231 int Lastcasetok; /* previous token within case */
232 int Lastcasetype; /* type of previous token within case */
233 int Switchflag = OFF; /* in switch = ON, otherwise OFF */
234
235
236 int Help = FALSE; /* help option flag */

```

```

237 int Assignments = TRUE; /* assignment option flag */
238 int Break_stmts = TRUE; /* breaks in switch/case option flag */
239 int Null_stmts = TRUE; /* unintentional null option flag */
240 int Pointers = TRUE; /* uninitialized pointers option flag */
241 int Scan_args = TRUE; /* improper scanf() args option flag */
242
243 char Command[50]; /* "system" command buffer */
244 char Workfile[14]; /* filename.c.p */
245
246 /*
247  * SYMBOL TABLE
248  */
249
250 struct tab {
251     char id[100];
252     char dctype;
253     char vartype;
254     int newtoken;
255     int block;
256     int init;
257 } Symtable[HASHSIZE], Nulltable;
258
259 FILE *inptr; /* file input pointer */
260
261 main (argc, argv)
262 int argc; /* argument counter */
263 char *argv[]; /* pointers to arguments */
264 {
265
266 /*
267  * INTEGER DECLARATIONS
268  */
269
270 int x; /* holds return from lex() */
271 int i; /* general index */
272 int token; /* value of token */
273 int init; /* pointer initialized flag */
274
275 /*
276  * CHARACTER DECLARATIONS
277  */
278
279 char *s; /* scratchpad pointer */
280 char vartype; /* variable type */
281 char name[100]; /* variable name */
282
283
284
285 * Get desired options - set option flags
286 */
287
288 while (argc > 1 && (*++argv)[0] != '-') {
289     for (s = argv[0]+1; *s != '\0'; s++) {
290         switch (*s) {
291             case 'h':
292                 Help = TRUE;
293                 break;
294             case 'a':
295                 Assignments = FALSE;
296                 break;

```

```

297         case 'b':
298             Break_stmts = FALSE;
299             break;
300         case 'n':
301             Null_stmts = FALSE;
302             break;
303         case 'p':
304             Pointers = FALSE;
305             break;
306         case 's':
307             Scan_args = FALSE;
308             break;
309         default:
310             fprintf(stdout, "Invalid Option '%c'\n", *s);
311             fprintf(stdout, "For help use: dust -h\n");
312             exit(0);
313     }
314 }
315 args--;
316 }
317
318 /*
319  * Does user want help? If so, print help info.
320  */
321 if (Help == TRUE) {
322     fprintf(stdout, "\nThis command searches C Language programs for a variety of errors.\n");
323     fprintf(stdout, "\nIt assumes your program has compiled successfully, but is not running properly.\n");
324     fprintf(stdout, "By default, most error checks are activated. To selectively\n");
325     fprintf(stdout, "suppress or activate checks, use the appropriate command line option(s).\n");
326     fprintf(stdout, "OPTIONS:\n");
327     fprintf(stdout, "\n-h HELP\n");
328     fprintf(stdout, "\n-a Suppress check for inappropriate assignments.\n");
329     fprintf(stdout, "\n-b Suppress check for breaks in switch/case statements.\n");
330     fprintf(stdout, "\n-n Suppress check for unintentional null statements.\n");
331     fprintf(stdout, "\n-p Suppress check for uninitialized pointers.\n");
332     fprintf(stdout, "\n-s Suppress check for improper scanf() function arguments.\n");
333     fprintf(stdout, "EXAMPLES:\n");
334     fprintf(stdout, "\t$ dust program.c\n");
335     fprintf(stdout, "\t$ dust -an program.c\n");
336     exit(1);
337 }
338
339 /* if no source file, print error */
340 if (argc != 2) {
341     fprintf(stdout, "You must specify a source file to be checked!\n");
342     fprintf(stdout, "For help use: dust -h\n");
343     exit(0);
344 }
345 else {
346     for (s = argv[0] + 1; *s != '\0'; s++)
347     {
348         /* if no .c suffix, print error */
349         if ((*s-1) != 'c' || (*s-2) != '.') {
350             printf("Source file name must end with '.c'\n");
351             exit(1);
352         }
353
354         /* if can't open source file, print error */
355         if ((inptr = fopen(argv[0], "r")) == NULL) {
356             fprintf(stderr, "Can't open '%s' for reading.\n", argv[0]);
357             exit(1);
358         }
359     }
360 }

```

```

357         fclose(inptr);
358     }
359     /* gather #include's in a file */
360     sprintf(Command, "grep -v '#include' %s > %s.x", argv[0], argv[0]);
361     system(Command);
362     sprintf(Workfile, "%s.x", argv[0]);
363     if((inptr = fopen(Workfile, "r")) == NULL) {
364         fprintf(stderr, "Can't open '%s' for reading\n", Workfile);
365         exit(1);
366     }
367     /* offset Linenum by number of #includes */
368     while ( (x = getch()) != EOF )
369         if (x == '\n') Linenum++;
370     fclose(inptr);
371     /* get rid of includes */
372     sprintf(Command, "grep -v '#include' %s > %s.x", argv[0], argv[0]);
373     system(Command);
374     /* run C preprocessor */
375     sprintf(Command, "cc -E %s.x > %s.p", argv[0], argv[0]);
376     system(Command);
377     /* get rid of superfluous output from cc -E */
378     sprintf(Command, "grep -v '^#' %s.p > %s.x", argv[0], argv[0]);
379     system(Command);
380     /* move file.c.x to file.c.p */
381     sprintf(Command, "mv %s.x %s.p", argv[0], argv[0]);
382     system(Command);
383     sprintf(Workfile, "%s.p", argv[0]);
384     if((inptr = fopen(Workfile, "r")) == NULL) {
385         fprintf(stderr, "Can't open %s for reading\n", Workfile);
386         exit(1);
387     }
388     while ( (x = lex()) != EOF ) { /* input tokens until EOF */
389         /* strip out comments */
390         if (Token == OPENCOMM && Type == OP) {
391             while(Token != CLSCOMM | Type != OP) {
392                 lex();
393             }
394         }
395         /* strip out quoted strings */
396         if ( (Token == DBLQT && Type == OP) &&
397             (Lastok != BKSLASH | Lastype != OP) ) {
398             do {
399                 lex();
400             }
401             while( (Token != DBLQT | Type != OP) &&
402                 (Lastok != BKSLASH && Lastype == OP) );
403             lex();
404         }
405     }
406     /*
407     * Determine what action to take for this input token.
408     */
409
410     if (Type == OP) {
411         switch(Token) {
412             /* if pending do, push brace onto Dostack */
413             case OPENBRACE:
414                 if (Dostack[0] != NULL) Dostack[Doptr++ ] = OPENBRACE;
415                 break;
416             /* clear out local variables after end of block */

```

```

417     case CLSBRAKE:
418         for ( i = 0; i < HASHSIZE; i++ ) {
419             if ( Syntable[i].block == Block)
420                 Syntable[i] = Nulltable;
421         }
422     /* if pending do, pop brace off Dostack */
423     if (Dostack[0] != NULL) Dostack[--Doptr] = NULL;
424     break;
425 }
426 }
427 else if ( Type == KEYWD ) {
428     switch (Token) {
429     case INT:
430     case LONG:
431     case SHORT:
432     case UNSIGNED:
433     case CHAR:
434     case FLOAT:
435     case DOUBLE:
436     case STATIC:
437     case REGISTER:
438         symhand();
439         /*dumpable();*/
440         break;
441     case IF:
442         if_while_hand(IF);
443         break;
444     case WHILE:
445         if_while_hand(WHILE);
446         break;
447     case DO:
448         if_while_hand(DO);
449         break;
450
451     case FOR:
452         for_hand(FOR);
453         break;
454
455     case SWITCH:
456         if ( Break_stmts == TRUE )
457             Switchflag = ON;
458         break;
459
460     case CASE:
461     case DEFAULT:
462         if ( Break_stmts == TRUE )
463             case_hand();
464         break;
465     case SCANF:
466     case $SCANF:
467     case PSCANF:
468         if ( Scan_args == TRUE )
469             scanf_hand();
470         break;
471
472     default:
473         continue;
474     }
475 }
476 }

```

```

477     else if (Type == ID && Pointers == TRUE) {
478         strcpy(name, Charbuff);
479         point_hand(Token, name);
480     }
481 }
482 } /* end of main */
483
484
485 /* DUMP SYMBOL TABLE - USED DURING DEVELOPMENT */
486
487 dumptable()
488 {
489     int i;
490     printf("\nSymtable\n");
491     for (i = 0; i < HASHSIZE; i++) {
492         if (Symtable[i].newtoken != 0)
493             printf("%d\tid = %s\tnewtoken = %d\tvartype = %d\tblock = %d\n",
1. Symtable[i].id, Symtable[i].newtoken, Symtable[i].vartype, Symtable[i].block);
494     }
495 }
496
497
498 /* LEXICAL ANALYZER - GET INPUT TOKENS */
499
500 lex()
501 {
502     Lastok = Token;
503     Lastype = Type;
504     if (Token != NL && Token != SEMI) {
505         Lastcasestok = Token;
506         Lastcasetype = Type;
507     }
508
509     for (Charpos = 0; Charpos < 100; Charpos++) {
510         Charbuff[Charpos] = 0;
511
512         Charpos = 0; /* reset Buffer index */
513         Type = 'x'; /* bogus initializer */
514
515         while ( ( Charbuff[Charpos] = getch() ) == ' ' (Charbuff[Charpos] == '\t') /* skip white space */
516             ; /* null statement */
517         if ( Charbuff[Charpos] == EOF ) {
518             fclose(inpfr);
519             sprintf(Command, "rm %s", Workfile);
520             system(Command);
521             exit(0);
522         }
523         if ( what_type(Charbuff[Charpos]) == LETTER )
524             key_id();
525         else {
526             switch(Charbuff[Charpos]) {
527                 case '0':
528                 case '1':
529                 case '2':
530                 case '3':
531                 case '4':
532                 case '5':
533                 case '6':
534                 case '7':
535                 case '8':

```

```

536     case '9':
537         numproc();
538         break;
539
540     case 'I':
541         exclamproc();
542         break;
543     case '%':
544         percentproc();
545         break;
546     case '&':
547         amperproc();
548         break;
549     case '(':
550         Type = OP;
551         Token = OPENPAR;
552         break;
553     case ')':
554         Type = OP;
555         Token = CLSPAR;
556         break;
557     case '*':
558         starproc();
559         break;
560     case '+':
561         plusproc();
562         break;
563     case ',':
564         Type = OP;
565         Token = COMMA;
566         break;
567     case '-':
568         minusproc();
569         break;
570     case '.':
571         Type = OP;
572         Token = DOT;
573         break;
574     case '/':
575         slashproc();
576         break;
577     case '<':
578         lessproc();
579         break;
580     case '=':
581         equalproc();
582         break;
583     case '>':
584         greatproc();
585         break;
586     case '?':
587         Type = OP;
588         Token = QUEST;
589         break;
590     case '[':
591         Type = OP;
592         Token = OPENBRACK;
593         break;
594     case ']':
595         Type = OP;

```

```

596         Token = CLSBRAK;
597         break;
598     case '"':
599         xortproc();
600         break;
601     case '?':
602         pipeproc();
603         break;
604     case "'":
605         Type = OP;
606         Token = TWOSCOMP;
607         break;
608     case '[':
609         Type = OP;
610         Token = OPENBRACE;
611         break;
612     case ']':
613         Type = OP;
614         Token = CLSBRAK;
615         break;
616     case ':':
617         Type = OP;
618         Token = SEMI;
619         break;
620     case ';':
621         Type = OP;
622         Token = COLON;
623         break;
624     case '\':
625         Type = OP;
626         Token = DBLOT;
627         break;
628     case '^':
629         Type = OP;
630         Token = SNGLOT;
631         break;
632     case '@':
633         Type = OP;
634         Token = POUND;
635         break;
636     case '\':
637         Type = OP;
638         Token = BKSLASH;
639         break;
640     case '~':
641         Type = OP;
642         Token = ATSIGN;
643         break;
644     case '`':
645         Type = OP;
646         Token = GRAVE;
647         break;
648     case '$':
649         Type = OP;
650         Token = DOLLAR;
651         break;
652     case '\n':
653         Type = OP;
654         Token = NL;
655         Linenum++;

```



```

656         break;
657     default:
658         printf("Illegal character - I quit!\n");
659         exit(-1);
660     }
661 } /* end of lex */
662
663
664 /*
665  * Determine whether token is a keyword or identifier.
666  */
667
668 key_id()
669 {
670     int c, n;
671     while ( what_type(c = Charbuff[ ++Charpos] = getch()) == LETTER ||
672            what_type(c) == DIGIT)
673     ;
674     ungetch(c);
675     Charbuff[Charpos] = '\0';
676     Charpos = 0;
677     if ( (n = findkey(Charbuff, keytable, NKEYS)) >= 0 ) {
678         Type = KEYWD;
679         Token = keytable[n].keynum;
680     }
681     else {
682         Type = ID;
683         Token = pchash(Charbuff);
684     }
685 } /* end of key_id */
686
687 /* bring in an input digit */
688
689 numproc()
690 {
691     int c;
692     while ( what_type(c = Charbuff[ ++Charpos] = getch()) == DIGIT)
693     ;
694     ungetch(c);
695     Charbuff[Charpos] = '\0';
696     Charpos = 0;
697     Type = 'c';
698     Token = atoi(Charbuff);
699 } /* end of numproc */
700
701
702
703 /*
704  * Use binary search to find keyword from table
705  */
706
707 findkey(word, tab, n)
708 char *word;
709 struct keys tab[NKEYS];
710 int n;
711 {
712     int low, high, mid, cond;
713     low = 0;
714     high = n - 1;
715     while (low <= high) {

```

```

716 mid = (low + high) / 2;
717 if ( (cond = strcmp(word, tab[mid] keyword)) < 0 )
718     high = mid - 1;
719 else if ( cond > 0 )
720     low = mid + 1;
721 else
722     return(mid);
723 }
724 return(-1);
725 } /* end of findkey */
726
727
728 /*
729  * get a character from input stream or from
730  * Buf[Bufp] if available.
731  */
732
733 getch()
734 {
735     if ( Bufp > 0 )
736         return (Buf[--Bufp]);
737     else
738         return(getc(inptr));
739 } /* end of getch */
740
741 /*
742  * put a character back 'on the shelf' in Buf[Bufp]
743  */
744
745 ungetch(c)
746 int c;
747 {
748     if (Bufp > BUFSIZE)
749         printf("ungetch: too many characters\n");
750     else
751         Buf[Bufp++ ] = c;
752 } /* end of ungetch */
753
754 /*
755  * Determine what type the token is ( Letter or digit )
756  * and return appropriate indication.
757  */
758
759 what_type(c)
760 int c;
761 {
762     if ( (c >= 'a' && c <= 'z') ||
763         (c >= 'A' && c <= 'Z') ||
764         (c == '.'))
765         return(LETTER);
766     else if (c >= '0' && c <= '9')
767         return(DIGIT);
768     else
769         return(c);
770 } /* end of what_type */
771
772 /*
773  * Process exclamation mark tokens.
774  */
775

```

```

776
777 exclamproc()
778 {
779     int c;
780     if ( (c = Charbuff[ ++Charpos ] = getch()) == '=' ) {
781         Type = OP;
782         Token = NE;
783         return;
784     }
785     else {
786         Type = OP;
787         Token = NOT;
788         ungetch(c);
789         Charbuff[Charpos] = '\0';
790         return;
791     }
792 } /* end of exclamproc */
793
794
795 /*
796  * Process percent sign tokens.
797  */
798
799 percentproc()
800 {
801     int c;
802     if ( (c = Charbuff[ ++Charpos ] = getch()) == '=' ) {
803         Type = OP;
804         Token = REMEQ;
805         return;
806     }
807     else {
808         Type = OP;
809         Token = REM;
810         ungetch(c);
811         Charbuff[Charpos] = '\0';
812         return;
813     }
814 } /* end of percentproc */
815
816
817 /*
818  * Process ampersand tokens
819  */
820
821 amperproc()
822 {
823     int c;
824     if ( (c = Charbuff[ ++Charpos ] = getch()) == '=' ) {
825         Type = OP;
826         Token = ANDEQ;
827         return;
828     }
829     else if (c == '&') {
830         Type = OP;
831         Token = LOGAND;
832     }
833     else {
834         Type = OP;
835         Token = AND;

```

```

836     ungetch(c);
837     Charbuff[Charpos] = '\0';
838     return;
839 }
840 } /* end of amperproc */
841
842
843 /*
844  * Process asterisk tokens.
845  */
846
847 starproc()
848 {
849     int c;
850     if ( (c = Charbuff[++Charpos] = getch()) == '=' ) {
851         Type = OP;
852         Token = STAREQ;
853         return;
854     }
855     else if (c == '/') {
856         Type = OP;
857         Token = CLSCOMM;
858     }
859     else {
860         Type = OP;
861         Token = STAR;
862         ungetch(c);
863         Charbuff[Charpos] = '\0';
864         return;
865     }
866 } /* end of starproc */
867
868
869 /*
870  * Process plus sign tokens.
871  */
872
873 plusproc()
874 {
875     int c;
876     if ( (c = Charbuff[++Charpos] = getch()) == '=' ) {
877         Type = OP;
878         Token = PLUSEQ;
879         return;
880     }
881     else if (c == '+') {
882         Type = OP;
883         Token = PPLUS;
884     }
885     else {
886         Type = OP;
887         Token = PLUS;
888         ungetch(c);
889         Charbuff[Charpos] = '\0';
890         return;
891     }
892 } /* end of plusproc */
893
894
895 /*

```

```

896  * Process minus sign tokens.
897  */
898
899  mil/minusproc()
900  {
901  int c;
902  if ( (c = Charbuff[++Charpos] = getch()) == '-' ) {
903      Type = OP;
904      Token = MINUSEQ;
905      return;
906  }
907  else if (c == '.') {
908      Type = OP;
909      Token = MMINUS;
910  }
911  else if (c == '>') {
912      Type = OP;
913      Token = ARROW;
914  }
915  else {
916      Type = OP;
917      Token = MINUS;
918      ungetch(c);
919      Charbuff[Charpos] = '\0';
920      return;
921  }
922  } /* end of minusproc */
923
924
925  /*
926  * Process slash tokens.
927  */
928
929  slashproc()
930  {
931  int c;
932  if ( (c = Charbuff[++Charpos] = getch()) == '/' ) {
933      Type = OP;
934      Token = SLASHEQ;
935      return;
936  }
937  else if (c == '*') {
938      Type = OP;
939      Token = OPENCOMM;
940  }
941  else {
942      Type = OP;
943      Token = SLASH;
944      ungetch(c);
945      Charbuff[Charpos] = '\0';
946      return;
947  }
948  } /* end of slashproc */
949
950
951  /*
952  * Process less than tokens.
953  */
954
955  lessproc()

```

```

956 {
957 int c;
958 if ( (c = Charbuff[ ++Charpos ] = getch()) == '=' ) {
959     Type = OP;
960     Token = LE;
961     return;
962 }
963 else if (c == '<') {
964     if ( (c = Charbuff[ ++Charpos ] = getch()) == '=' ) {
965         Type = OP;
966         Token = SHFTLEQ;
967         return;
968     }
969     else {
970         Type = OP;
971         Token = SHFTL;
972         ungetch(c);
973         Charbuff[Charpos] = '\0';
974         return;
975     }
976 }
977 else {
978     Type = OP;
979     Token = LT;
980     ungetch(c);
981     Charbuff[Charpos] = '\0';
982     return;
983 }
984 } /* end of lessproc */
985
986
987 /*
988  * Process equal sign tokens.
989  */
990
991 equalproc()
992 {
993 int c;
994 if ( (c = Charbuff[ ++Charpos ] = getch()) == '=' ) {
995     Type = OP;
996     Token = EOEO;
997     return;
998 }
999 else {
1000     Type = OP;
1001     Token = EQ;
1002     ungetch(c);
1003     Charbuff[Charpos] = '\0';
1004     return;
1005 }
1006 } /* end of equalproc */
1007
1008
1009 /*
1010  * Process greater than tokens
1011  */
1012
1013 greatproc()
1014 {
1015 int c;

```

```

1016 if ( (c = Charbuff[++Charpos] = getch()) == '=' ) {
1017     Type = OP;
1018     Token = LE;
1019     return;
1020 }
1021 else if (c == '>') {
1022     if ( (c = Charbuff[++Charpos] = getch()) == '=' ) {
1023         Type = OP;
1024         Token = SHFTREQ;
1025         return;
1026     }
1027     else {
1028         Type = OP;
1029         Token = SHFTR;
1030         ungetch(c);
1031         Charbuff[Charpos] = '\0';
1032         return;
1033     }
1034 }
1035 else {
1036     Type = OP;
1037     Token = GT;
1038     ungetch(c);
1039     Charbuff[Charpos] = '\0';
1040     return;
1041 }
1042 } /* end of greatproc */
1043
1044
1045 /*
1046  * Process xor tokens
1047  */
1048
1049 xorproc()
1050 {
1051     int c;
1052     if ( (c = Charbuff[++Charpos] = getch()) == '=' ) {
1053         Type = OP;
1054         Token = XOREQ;
1055         return;
1056     }
1057     else {
1058         Type = OP;
1059         Token = XOR;
1060         ungetch(c);
1061         Charbuff[Charpos] = '\0';
1062         return;
1063     }
1064 } /* end of xorproc */
1065
1066
1067 /*
1068  * Process pipe symbol tokens.
1069  */
1070
1071 pipeproc()
1072 {
1073     int c;
1074     if ( (c = Charbuff[++Charpos] = getch()) == '=' ) {
1075         Type = OP;

```

```

1076     Token = OREQ;
1077     return;
1078 }
1079 else if (c == '?') {
1080     Type = OP;
1081     Token = LOGOR;
1082 }
1083 else {
1084     Type = OP;
1085     Token = OR;
1086     ungetch(c);
1087     Charbuff[Charpos] = '\0';
1088     return;
1089 }
1090 } /* end of pipeproc */
1091
1092
1093 /*
1094  * Hashing function
1095  */
1096
1097 pchash(s)
1098 char *s;
1099 {
1100     int hashval;
1101     for ( hashval = 0; *s != '\0'; )
1102         hashval += *s++;
1103     return (hashval % HASHSIZE);
1104 }
1105
1106 /*
1107  * Handles symbol table entries
1108  */
1109
1110 symhand()
1111 {
1112     char vartype;
1113     int token;
1114     char name[BUFSIZE];
1115     int init = NO;
1116     lex();
1117     while ( Token != SEMI ) {
1118         if ( Token == STAR ) {
1119             lex();
1120             token = Token;
1121             strcpy(name, Charbuff);
1122             if ( Token == STAR ) {
1123                 vartype = PTRARRAY;
1124                 lex();
1125                 token = Token;
1126                 strcpy(name, Charbuff);
1127                 make_entry(vartype, token, name, init);
1128             }
1129             else {
1130                 lex();
1131                 if ( Token == OPENBRK ) {
1132                     vartype = PTRARRAY;
1133                     make_entry(vartype, token, name, init);
1134                 }
1135                 else {

```



```

1136     vartype = PTR;
1137     if (Token == EQ && Type == OP ) init = YES;
1138     make_entry(vartype, token, name, init);
1139 }
1140 }
1141 }
1142 else if ( Type == ID ) {
1143     token = Token;
1144     strcpy(name, Charbuff);
1145     lex();
1146     if ( Token == OPENBRAC ) {
1147         vartype = ARRAY;
1148         make_entry(vartype, token, name, init);
1149     }
1150     else {
1151         vartype = SINGLE;
1152         make_entry(vartype, token, name, init);
1153     }
1154 }
1155 if ( Token == SEMI )
1156     break;
1157 lex();
1158 }
1159 } /* end of symhand */
1160
1161 /*
1162  * Makes symbol table entries
1163  */
1164
1165 make_entry(vartype, token, name, init)
1166 char vartype;
1167 int token;
1168 char *name;
1169 int init;
1170 {
1171     int search;
1172     int count = 0;
1173     search = token;
1174     while (Symtable[search++ % HASHSIZE].newtoken != 0 && count <= HASHSIZE)
1175         ;
1176     if (count > HASHSIZE) {
1177         printf("Symbol table overflow - I quit!\n");
1178         exit(1);
1179     }
1180     search--;
1181     Symtable[search].newtoken = search;
1182     strcpy(Symtable[search].id, name);
1183     Symtable[search].block = Block;
1184     Symtable[search].vartype = vartype;
1185     Symtable[search].init = init;
1186 } /* end of make_entry */
1187
1188 /*
1189  * Finds symbol table entries
1190  */
1191
1192 find_entry(vartype, token, name, init)
1193 char *vartype;

```

```

1196 int token;
1197 char *name;
1198 int *init;
1199 {
1200 int count;
1201 int search = token;
1202 int searchblock;
1203 if ( (token == Symtable[token].newtoken) &&
1204      (strcmp(name, Symtable[token].id) == 0) &&
1205      (Block == Symtable[search].block) )
1206 {
1207     *vartype = Symtable[search].vartype;
1208     *init = Symtable[search].init;
1209     return(Symtable[search].newtoken);
1210 }
1211 else {
1212     for (searchblock = Block; searchblock >= 0; searchblock--) {
1213         count = 0;
1214         search = token + 1;
1215         while ( ((strcmp(name, Symtable[search % HASHSIZE].id) != 0)
1216                 (Block != Symtable[search % HASHSIZE].block)) &&
1217                 (search <= HASHSIZE) ) {
1218             count++;
1219             search++;
1220         }
1221     }
1222 }
1223 if (strcmp(name, Symtable[search % HASHSIZE].id) == 0) {
1224     *vartype = Symtable[search].vartype;
1225     *init = Symtable[search].init;
1226 }
1227 else {
1228     *init = 1;
1229 }
1230 return(Symtable[search].newtoken);
1231 } /* end of find_entry */
1232
1233
1234 /*
1235  * Process if or while statement
1236  */
1237
1238 if_while_hand(condtype)
1239 int condtype;
1240 {
1241 int tokbuf[BUFSIZE];
1242 char typebuf[BUFSIZE];
1243 int i;
1244 int parent = 0;
1245 for ( i = 0; i < 100; i++ ) {
1246     tokbuf[i] = 0;
1247     typebuf[i] = '\0';
1248 }
1249 i = 0;
1250 if (condtype == DO) {
1251     Dostack[Doptr++] = DO;
1252     return;
1253 }
1254 lex();
1255 if ( (typebuf[i] == Type. tokbuf[i] == Token) != OPENPAR || Type != OP)

```

```

1256 printf("line %3d: - No \" after if/while!\n",Linenum);
1257 else parent++;
1258
1259 i++;
1260 while ( parent > 0 ) {
1261     lex();
1262     typebuf[i] = Type;
1263     tokbuf[i++] = Token;
1264     if ( Type == OP )
1265         switch (Token) {
1266             case OPENPAR:
1267                 parent++;
1268                 break;
1269             case CLSPAR:
1270                 parent--;
1271                 break;
1272         }
1273     }
1274     if ( Assignments == TRUE )
1275         cond_hand(tokbuf, typebuf, condtype);
1276     if ( Null_stmts == TRUE )
1277         null_hand(condtype);
1278 } /* end of if_while_hand */
1279
1280
1281
1282 /*
1283  * Process for statement
1284  */
1285
1286 for_hand(condtype)
1287 int condtype;
1288 {
1289     int tokbuf[BUFSIZE];
1290     char typebuf[BUFSIZE];
1291     int i;
1292     for ( i = 0; i < 100; i++ ) {
1293         tokbuf[i] = 0;
1294         typebuf[i] = '\0';
1295     }
1296     i = 0;
1297     while ( Token != SEMI / Type != OP ) {
1298         lex();
1299     }
1300     typebuf[i] = OP;
1301     tokbuf[i++] = OPENPAR;
1302     Token = NULL;
1303     while ( Token != SEMI / Type != OP ) {
1304         lex();
1305         typebuf[i] = Type;
1306         tokbuf[i++] = Token;
1307     }
1308     typebuf[i] = OP;
1309     tokbuf[i++] = CLSPAR;
1310     if ( Assignments == TRUE )
1311         cond_hand(tokbuf, typebuf, condtype);
1312     if ( Null_stmts == TRUE )
1313         null_hand(condtype);
1314 } /* end of for_hand */
1315

```

```

1316
1317 /*
1318  * Process conditional part of if, while, for, do-while
1319  */
1320
1321 cond_hand(tokbuf, typebuf, condtype)
1322 int *tokbuf;
1323 char *typebuf;
1324 int condtype;
1325 {
1326     int i;
1327     int assignment = 0;
1328     int relopent = 0;
1329     int logopent = 0;
1330     int parentent = 0;
1331     int optype[BUFSIZE];
1332
1333     for ( i = 0; i <= BUFSIZE; i++ ) optype[i] = 0;
1334
1335     for ( i = 0; (tokbuf[i] != 0)  (typebuf[i] == CONST); i++ ) {
1336         if (typebuf[i] == OP) {
1337             switch(tokbuf[i]) {
1338                 case EQ:
1339                     if ( tokbuf[i-1] != SNGLEQT ) {
1340                         assignment += 1;
1341                         optype[i] = ASSIGN;
1342                     }
1343                     break;
1344
1345                 case NE:
1346                 case LT:
1347                 case LE:
1348                 case EQEQ:
1349                 case GT:
1350                 case GE:
1351                     relopent += 1;
1352                     optype[i] = RELOP;
1353                     break;
1354
1355                 case LOGOR:
1356                 case LOGAND:
1357                     logopent += 1;
1358                     optype[i] = LOGOP;
1359                     break;
1360             }
1361         }
1362     }
1363
1364     i = 0;
1365     if ( relopent == 0 && assignment == 0 )
1366         printf( "line %3d: - No relational operators in \"%s\".\n", Lnum, keytable[condtype].keyword);
1367     else if ( relopent == 0 && assignment > 0 ) {
1368         if ( strcmp( keytable[condtype].keyword, "for" ) == 0 )
1369             printf( "line %3d: - Misuse of assignment operator ( = ) in \"%s\" ( try <= ),\n",
1370                 Lnum, keytable[condtype].keyword );
1371         else
1372             printf( "line %3d: - Misuse of assignment operator ( = ) in \"%s\" ( try = ),\n",
1373                 Lnum, keytable[condtype].keyword );
1374     }
1375     else if ( relopent == logopent && assignment > 0 ) {

```

```

1374 if (strcmp(keytable[cndtype] keyword, "for") == 0)
1375 printf("line %3d: - Misuse of assignment operator ( = ) in \"%s\" ( try < = ).\n",
Lineum = , keytable[cndtype] keyword);
1376 else
1377 printf("line %3d: - Misuse of assignment operator ( = ) in \"%s\" ( try == ).\n",
Lineum, keytable[cndtype] keyword);
1378 }
1379 else if ( relopent > 0 && assignment > 0 ) {
1380 i = 0;
1381 while ( assignment > 0 && relopent > 0 ) {
1382 if (typebuf[i] == OP ) {
1383 switch(optype[i]) {
1384 case ASSIGN:
1385 i++;
1386 if (relopent > 0) {
1387 while ( optype[i] != RELOP ) {
1388 if ( tokbuf[i] == OPENPAR && typebuf[i] == OP ) parentnt--;
1389 if ( tokbuf[i] == CLSPAR && typebuf[i] == OP ) parentnt++;
1390 if ( tokbuf[i] == EQ && typebuf[i] == OP ) assignment--;
1391 i++;
1392 }
1393 if ( parentnt <= 0 )
1394 printf("line %3d: - Operator precedence error involving assignment in \"%s\".\n",
Lineum, keytable[cndtype] keyword);
1395 }
1396 parentnt = 0;
1397 assignment--;
1398 relopent--;
1399 break;
1400
1401 case RELOP:
1402 i++;
1403 if ( assignment > 0 ) {
1404 while ( ntype[i] != ASSIGN ) {
1405 if ( tokbuf[i] == OPENPAR && typebuf[i] == OP ) parentnt++;
1406 if ( tokbuf[i] == CLSPAR && typebuf[i] == OP ) parentnt--;
1407 i++;
1408 }
1409 if ( parentnt <= 0 )
1410 printf("line %3d: - Operator precedence error involving assignment in \"%s\".\n",
Lineum, keytable[cndtype] keyword);
1411 }
1412 parentnt = 0;
1413 assignment--;
1414 relopent--;
1415 break;
1416
1417 default:
1418 i++;
1419 }
1420 }
1421 else if (typebuf[i] == ID )
1422 i++; /* do nothing (call pointer checker later) */
1423 else if (typebuf[i] == KEYWD )
1424 i++; /* do nothing (call scanf checker later) */
1425 }
1426 }
1427 }
1428
1429

```

```

1430  * Look for null statement in wrong place after if, while, for.
1431  */
1432
1433  null_hand(condtype)
1434  int condtype;
1435  {
1436      int loopcnt = 0;
1437      if ( condtype == FOR ) {
1438          while ( Token != CLSPAR | Type != OP )
1439              lex();
1440      }
1441      lex();
1442      while ( (Token != NL) && ( Token != SEMI) | Type != OP ) {
1443          loopcnt++;
1444          lex();
1445      }
1446      if ( Token == NL && Type == OP ) loopcnt++;
1447      switch(condtype) {
1448
1449          case WHILE:
1450              if (loopcnt == 0 && Dostack[Dopr - 1] != DO)
1451                  printf("line %3d: - Null statement (,) after \"%s\".\n", Lnumum, keytable[condtype].keyword);
1452              else if (loopcnt == 0 && Dostack[Dopr - 1] == DO)
1453                  Dostack[--Dopr] = NULL;
1454              break;
1455
1456          case IF:
1457          case FOR:
1458              if (loopcnt == 0)
1459                  printf("line %3d: - Null statement (,) after \"%s\".\n", Lnumum, keytable[condtype].keyword);
1460              break;
1461      }
1462      /* end of null_hand */
1463
1464      /*
1465       * Look for missing break statements in case constructs.
1466       */
1467
1468      case_hand()
1469      {
1470          if (Lastcasetok == COLON | Lastcasetok == BREAK | Switchflag == ON)
1471              Switchflag = OFF;
1472          else
1473              printf("line %3d: - No break at end of \"%case\".\n", Lnumum - 1);
1474      } /* end of case_hand */
1475
1476      /*
1477       * Process scanf() statement
1478       */
1479
1480      scanf_hand()
1481      {
1482          int tokbuf[BUFSIZE];
1483          int qtent = 0;
1484          char typebuf[BUFSIZE];
1485          char vartype;
1486          char name[100];
1487          int i;
1488          int i;
1489      }

```

```

1490 for ( i = 0; i < 100; i++ ) {
1491     tokbuf[i] = 0;
1492     typebuf[i] = '\0';
1493 }
1494 i = 0;
1495 lex();
1496 while ( qcnt < 2 ) {
1497     if (Token == DBLQT && Type == OP && Lastok != BKSLASH)
1498         qcnt++;
1499     lex();
1500 }
1501 if ( Token != COMMA ) printf("Error in scanf format!\n");
1502 lex();
1503 if ( Type == ID ) strcpy(name, Charbuff);
1504 while ( ! (Token == CLSPAR && Type == OP) ) {
1505     while ( ! ( (Token == COMMA && Type == OP) / (Token == CLSPAR && Type == OP) ) ) {
1506         tokbuf[i] = Token;
1507         typebuf[i++] = Type;
1508         lex();
1509         if ( Type == ID ) strcpy(name, Charbuff);
1510     }
1511     i = 0;
1512     while ( typebuf[i] != ID )
1513         i++;
1514     find_entry(&vartype, tokbuf[i], name, &init);
1515     if (vartype == SINGLE && (tokbuf[0] != AND : typebuf[0] != OP) )
1516         printf("line %3d: Incorrect address specification for \"%s\" in scanf()", initentry \"%s\" '\n',
1517             Linenum, name, name);
1518     else if ( vartype == ARRAY && !( (i == 0 && tokbuf[1] != OPENBRAK)
1519         | (tokbuf[0] == AND && tokbuf[2] == OPENBRAK) ) )
1520         printf("line %3d: Incorrect address specification for \"%s\" in scanf()", initentry \"%s\" or \"%s[n]\" '\n',
1521             Linenum, name, name, name);
1522     else if ( vartype == PTR ) {
1523         if ( i != 0 : typebuf[0] != ID )
1524             printf("line %3d: Incorrect address specification for \"%s\" in scanf()", initentry \"%s\" '\n',
1525                 Linenum, name, name);
1526         else
1527             if ( Pointers == TRUE ) point_hand(tokbuf[0], name);
1528     }
1529     else if ( vartype == PTRARRAY && !( (i == 0 && typebuf[0] == ID)
1530         | (i == 0 && tokbuf[1] == OPENBRAK)
1531         | (tokbuf[0] == STAR && typebuf[1] == ID
1532         && tokbuf[2] != OPENBRAK) ) )
1533         printf("line %3d: Incorrect address specification for \"%s\" in scanf()", initentry \"%s\" or \"%s[n]\" '\n',
1534             Linenum, name, name, name);
1535     for ( i = 0; i < 100; i++ ) {
1536         tokbuf[i] = 0;
1537         typebuf[i] = '\0';
1538     }
1539     i = 0;
1540     if ( Token == CLSPAR && Type == OP ) break;
1541     lex();
1542     if ( Type == ID ) strcpy(name, Charbuff);
1543 }
1544 /* end of scanf_hand */
1545
1546 /*
1547 * Look for uninitialized pointers
1548 */

```

```

1546
1547 point_hand(token,name)
1548 int token;
1549 char *name;
1550 {
1551 char vartype;
1552 int init;
1553 find_entry(&vartype, token, name, &init);
1554 if ( vartype == PTR ) {
1555     if ( init == NO ) {
1556         lex();
1557         if ( Token == EQ ) {
1558             set_init(token, name);
1559         }
1560         else
1561             printf("line %3d: Possible uninitialized pointer - \"%s\".\n",Linenum, name);
1562     }
1563 }
1564 else {
1565     return;
1566 }
1567 } /* end of point_hand */
1568
1569
1570 /*
1571  * set initialized flag to YES
1572  */
1573
1574 set_init(token,name)
1575 int token;
1576 char *name;
1577 {
1578 int count;
1579 int search = token;
1580 int searchblock;
1581 if ( (token == Symtable[token].newtoken) &&
1582     (strcmp(name, Symtable[token].id) == 0) &&
1583     (Block == Symtable[search].block) )
1584 {
1585     Symtable[search].init = YES;
1586     return(Symtable[search].newtoken);
1587 }
1588 else {
1589     for (searchblock = Block; searchblock >= 0; searchblock--) {
1590         count = 0;
1591         search = token + 1;
1592         while ( ((strcmp(name, Symtable[search % HASHSIZE].id) != 0) ||
1593             (Block != Symtable[search % HASHSIZE].block)) &&
1594             (count <= HASHSIZE) ) {
1595             count ++;
1596             search ++;
1597         }
1598     }
1599 }
1600 Symtable[search].init = YES;
1601 return(Symtable[search].newtoken);
1602 } /* end of set_init */
1603
1604 /* end of program */

```


ANALYZING "C" PROGRAMS FOR COMMON ERRORS

by

Dennis M. Frederick

B.S., University of Missouri, 1970

AN ABSTRACT OF A REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

COMPUTER SCIENCE

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

ANALYZING "C" PROGRAMS FOR COMMON ERRORS

ABSTRACT

Several programming errors commonly made by users of the C Programming Language escape detection by the C compiler. These errors in usage result in statements which are syntactically and semantically correct, but which are usually not what the programmer intended and cause incorrect program execution. Several debugging tools are available for C Language, but none of them detect these commonly made errors. The focus of this investigation is the development of a tool to analyze C programs to detect and report these errors. Specifically, it detects inappropriate uses of the assignment operator and operator precedence errors in control constructs, unintended null statements, omission of break statements from switch/case constructs, improper address specifications in scanf() function calls, and uninitialized pointers.